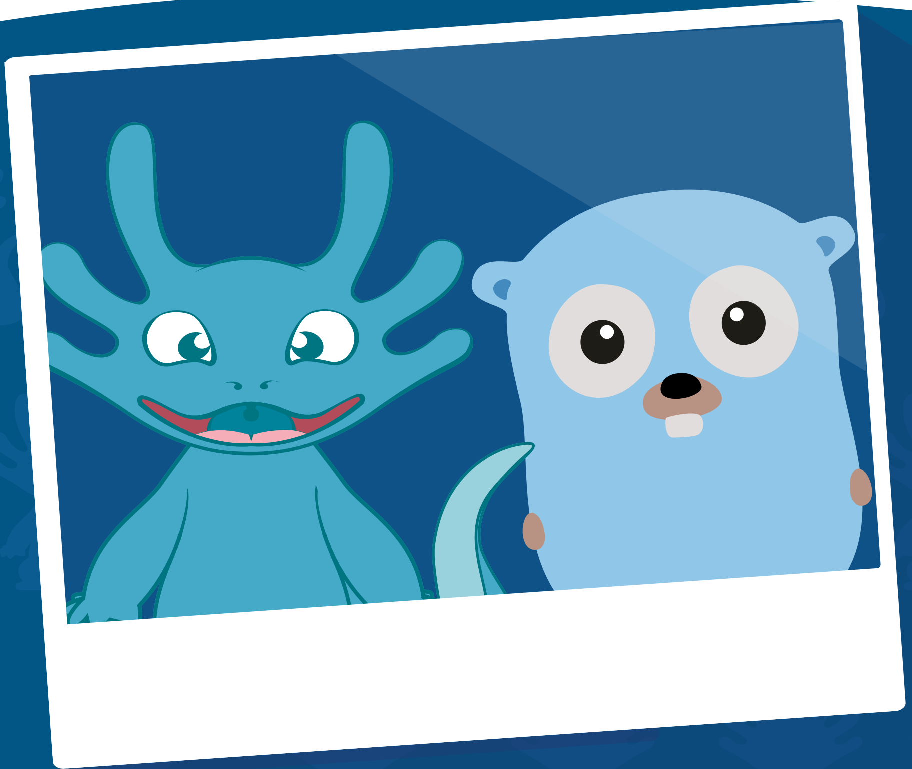
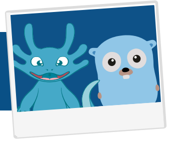
Comprendre GO



[sf≡ir]

Gopher-front.{ai,svg,png} was created by Takuya Ueda (https://twitter.com/tenntenn).
Licensed under the Creative Commons 3.0 Attributions license.

Table des matières



Pourquoi Go ? 5

À propos de ce document 6

Un peu d'histoire 7

Chapitre 1. Mon premier programme Go 8
Installation 9
Tooling et IDEs 10
Hello, Go 11
Workspace et packages 11
"go build et "go install" 13
Exécutable vs. bibliothèque 13

Chapitre 2. Les bases du langage 16
Les types en Go 18
  Conversions 19
Itérations et conditionnelles 19
Structures 22
Tableaux et *slices* 23
Le type *map* 24
Les subtilités des chaînes de caractères 25
Pointeurs 27
Type "fonction" 28
Méthodes 30
  Go et la composition 32
Interface 34
Tags et réflexion 35

Chapitre 3. Testing 37
Assertions 39

Chapitre 4. Concurrence 40
Goroutines 41
Channels 42

Chapitre 5. Accès aux données 45
MongoDB 46
SQL 48
Package "sql" 48
ORM 50

Chapitre 6. Web 52
Basics avec http 53
Autres options 55
Multiplexeurs 55
Middleware 55
Frameworks web 57
Tests web 59

Chapitre 7. Conteneurisation 60

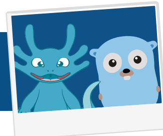
Conclusion 62

Aller plus loin 63
Livres 63
Ressources online 63
Blogs et sites consacrés à Go 63
Articles et tutoriels 64
Outils et bibliothèques 64
Vidéos 64

À propos de SFEIR 67

Remerciements 70

Pourquoi Go ?



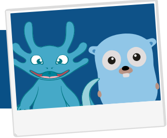
Personne ne peut nier que les langages compilés et fortement typés ont des avantages énormes : rapidité d'exécution, vérifications statiques faites par le compilateur, complétion pertinente dans l'IDE, etc. Oui mais la compilation… c'est lent.

Du coup vive les langages interprétés ! Le scripting c'est fantastique, on modifie le code et on recharge, c'est instantané… Génial ! Mais du coup la performance en prend un coup, adieu les vérifications statiques et la complétion pertinente.

Et s'il y avait une troisième voie ? Go est à la fois fortement typé et compilé, mais son compilateur est hyper rapide et conçu pour faire oublier qu'il n'est pas interprété. Go compile en natif (pour de nombreuses plateformes) et pas en bytecode, ce qui le rend d'autant plus performant à l'exécution; un programme Go est performant dès le démarrage et n'a pas besoin de warm-up pour que la machine virtuelle optimise son comportement. Un binaire Go est auto-suffisant et embarque toutes ses dépendances, le rendant facile à distribuer et containeriser.

Et si Go était votre prochain langage ?

À propos de ce document



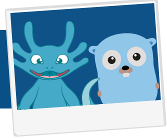
Commençons par ce que ce document n'est pas : il ne s'agit pas d'une formation au langage Go et à son utilisation. Même si nous espérons que cela puisse vous fournir les bases, il existe des formations dédiée, par exemple la SFEIR School "Go Lang 200" : https://www.sfeir.com/formation/school/sgo200

Il s'agit ici de vous donner un aperçu de ce qu'est le langage Go, pas seulement de sa syntaxe et de ses outils, mais aussi de sa philosophie et de ses partis-pris, de son écosystème et des bibliothèques disponibles. Le but ultime est que vous vous fassiez une opinion personnelle pour répondre à la question : est-ce que Go est fait pour moi ? Est-ce qu'il peut m'aider à répondre à mes problématiques ?

Comme pour un développeur le code est essentiel, j'ai essayé d'inclure des exemples de code Go partout où c'était utile, car c'est la première approche d'un langage et c'est en lisant du code qu'on peut en avoir le premier "feeling". Beaucoup de développeurs "back" viennent de Java ou possèdent un minimum de culture Java, quand c'était utile j'ai donc mis en parallèle les deux univers.

Enfin nous espérons que cela vous donnera au moins l'envie de vous y mettre et d'essayer Go vous-même, car c'est encore la meilleure façon d'explorer un nouveau langage !

Un peu d'histoire



Go est le résultat d'une réflexion commencée en 2007 par les ingénieurs de Google : Robert Griesemer, Rob Pike, et Ken Thompson (ce dernier ayant grandement collaboré au développement d'Unix et du langage C, rien que ça), motivés par la conscience de la nécessité d'un meilleur outil pour ce qu'ils faisaient chez la firme de Mountain View. L'objectif est devenu la création d'un nouveau langage qui garderait les caractéristiques positives des langages existants, tout en corrigeant leurs défauts (particulièrement C++).

Selon Robert Griesemer, les principes qui ont guidé sa conception étaient les suivants :
- Simplicité, sécurité et lisibilité,
- Orthogonalité de la conception (absence d'interdépendance),
- Minimalisme,
- Centré sur l'expression d'algorithmes (pas sur le typage),
- Conscience claire de ce qu'il fallait ne pas faire !

D'autres principes ont façonné Go :
- Typage statique,
- Utilisable sans IDE,
- Support natif du réseau et du parallélisme.

Il en a résulté un langage qui empruntait au Pascal pour la structure, au C pour la syntaxe, à smalltalk pour l'orientation objet et de nombreuses autres inspirations.

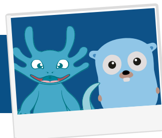
Go prend aussi des partis-pris radicaux sur certains aspects :
- Pas d'héritage : Go incite à la composition et possède un système d'interface implicite (un type implémente une interface si toutes les méthodes de l'interface sont définies pour ce type).
- Pas de warning : soit ça compile, soit ça ne compile pas ! Par exemple déclarer une variable ou un import et ne pas l'utiliser est une erreur fatale en Go.
- Go possède un style de formatage standard pour le code (indentation, espacement, etc.).
- Pas de types génériques (mais ce point est sujet à discussion).

Aujourd'hui de nombreux projets et compagnies utilisent Go, parmi lesquels Google évidemment, mais aussi Docker, Kubernetes, OpenShift, CoreOS, Couchbase, CloudFoundry, Dropbox, Netflix, SoundCloud, Uber, Hashicorp, etc.

Chapitre 1

Mon premier programme Go

Mon premier programme Go



Installation

Pour démarrer il vous faudra une distribution Go.
Go est officiellement supporté sur les plateformes suivantes :

OS	Architecture	Remarques
FreeBSD 10.3 ou ultérieur	amd64, 386	Debian GNU/kFreeBSD non supporté
Linux 2.6.23 ou ultérieur avec glibc	amd64, 386, arm, arm64, s390x, ppc64le	CentOS/RHEL 5.x non supporté
Mac OS X 10.8 ou ultérieur	amd64	
Windows XP SP2 ou ultérieur	amd64, 386	

A noter que Go sait également faire de la **cross-compilation** pour cibler d'autres plate-formes, quelle que soit la machine hôte du compilateur (utile pour cibler les devices type Raspberry Pi).

De manière générale, l'installation consiste en :
1. Récupérer l'archive (zip ou tar.gz) correspondant à la plateforme cible depuis la page de téléchargement http://golang.org/dl
2. Extraire l'archive à l'endroit souhaité (standard : `/usr/local/go` pour les Unix-like, `C:\go` pour Windows).
3. Si le répertoire d'installation n'est pas le répertoire standard, définir une variable d'environnement `GOROOT` qui désigne le répertoire d'installation.
4. Ajouter le répertoire `go/bin` (`/usr/local/go/bin`, `C:\Go\bin` ou autre selon le répertoire d'installation) au PATH du système.
5. C'est tout !

Si vous préférez un installeur interactif, il existe un `.pkg` pour MacOS et un `.MSI` pour Windows disponible également sur la page de download.

Sous windows il est également possible de l'installer via *chocolatey* : `choco install golang`, ou *scoop* : `scoop install go`; de même sous MacOS il est possible de l'installer via *brew* : `brew install golang`.

Pour vérifier l'installation, ouvrez un *shell* ou interpréteur de commandes Windows et tapez “go version”, si l’installation est correcte Go doit répondre avec une ligne similaire à ceci :

```
$ go version
go version go1.10.2 darwin/amd64
```

Note : dans la suite les exemples de lignes de commande sont orientés Linux/MacOS mais se transposent facilement sous Windows.

Tooling et IDEs

Comme un des objectifs de la philosophie initiale de Go est de réduire le tooling, il n'existe pas (du moins pas encore) une galaxie d'outils comme on peut trouver sur d'autres plateformes : outils de build, gestion de dépendances (même si *dep* a fait récemment son apparition), pré/post-processing de code, instrumentation, etc. Dans beaucoup de cas, le SDK Go se suffit à lui-même, le processus de build étant inclus (on en reparlera plus loin).

Cependant, il existe heureusement des IDEs ou éditeurs spécialisés qui facilitent le développement en Go.

Citons (sans ordre spécifique) :
- **IntelliJ** propose un plugin dédié à Go (nécessite la version *Ultimate* d'IntelliJ),
- Jetbrains (l’éditeur d’IntelliJ) propose également un IDE (sur base d’IntelliJ et du plugin Go) entièrement dédié à Go, qui s’appelle “**GoLand**” : http://www.jetbrains.com/go/download,
- **GoClipse** est une déclinaison d’Eclipse destinée au développement Go : http://github.com/GoClipse/goclipse,
- **LiteIDE** est un IDE cross-platform spécifiquement développé pour Go : http://github.com/visualfc/liteide,
- La plupart des éditeurs de texte (**Visual Studio Code**, **SublimeText**, **Atom**, **vim** pour les principaux) supportent un mode de *syntax-highlighting* pour Go, voire des fonctionnalités orientées développement. A noter que le plugin Visual Studio Code est soutenu par Microsoft et s'appuie sur les outils standard (goimport, go vet, ...).

L'important est de faire son choix en fonction de ses habitudes et de l'environnement dans lequel on est le plus à l'aise.

Hello, Go

Impossible de commencer avec un langage sans sacrifier à la tradition du “Hello world”...

Créez un répertoire `hello` à l’endroit de votre choix; dans ce répertoire créez un fichier `hello.go` avec le contenu suivant :

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, Go")
}
```

Placez vous dans ce répertoire et tapez :

```
$ go build hello.go
```

Si tout s’est bien passé, la commande rend la main sans rien écrire. C’est un peu perturbant mais de façon générale, la commande “go” n’écrit de message qu’en cas d’erreur. Cela correspond à la volonté de concision de ses créateurs : Go ne parle pas pour ne rien dire !

Le résultat de la commande est un fichier exécutable `hello` (`hello.exe` sous Windows) dans le répertoire courant. Si vous l’exécutez, le résultat est sans surprise :

```
$ ./hello
Hello, Go
```

Workspace et packages

Le développement avec Go se passe dans un *workspace*.

Un workspace Go est un répertoire qui contient les sous-répertoires suivants :
- `src` : le code source,
- `pkg` : les bibliothèques (binaires compilés non exécutables),
- `bin` : les exécutables.

Par défaut, depuis la version 1.8, votre workspace se trouve dans `$HOME/go` (`%userprofile%\go` sous Windows); c’est là que la commande `go` ira chercher les différents fichiers nécessaires à la compilation. Vous pouvez toutefois placer votre workspace où vous le souhaitez, ou basculer d’un workspace à un autre, en positionnant la variable d’environnement `GOPATH`.

Tip : c'est une bonne pratique d'ajouter également `$GOPATH/bin` à votre `PATH` pour pouvoir appeler les exécutables Go faisant partie du workspace depuis n'importe quel répertoire courant.

À la différence d'autres langages pour lesquels chaque projet constitue un workspace, un projet Go correspond à un sous-répertoire de `src` (y-compris à plusieurs niveau) dans un workspace. Le chemin complet qui permet d'accéder aux sources à partir de `src` constitue un "package" pour Go.

Exemple : si vous mettez vos sources Go dans `$GOPATH/src/hello/world`, cela définit de facto un package "hello/world". Lorsque vous compilez ce package (avec "`go install`" qu'on verra plus loin), les fichiers résultants seront placés :
- dans `pkg/<archi>/hello/world` (où <archi> correspond à l'architecture de la plateforme, par exemple "`darwin_amd64`" pour MacOS) : le ou les binaires non exécutables,
- dans `bin` l'exécutable final (le cas échéant).

Pour travailler avec des sources Go sous contrôle de version (par ex. Git), il est d'usage de placer la copie locale du projet sous le répertoire `src`, en calquant la structure de répertoires sur l'URL du projet.

Exemple : si vous avez un projet sous GitHub avec l'URL `https://github.com/ogerardin/go-playground`, il devrait être placé dans votre Workspace Go dans `$GOPATH/src/github.com/ogerardin/go-playground;` et le chemin du package serait alors "`github.com/ogerardin/go-playground`".

Si le projet comprend lui-même des sous-répertoires, ils s'ajoutent naturellement pour désigner le sous-package, par exemple si les sources sont `https://github.com/ogerardin/go-playground/hello/world`, le chemin du package est alors (toujours à partir de `src`) "`github.com/ogerardin/go-playground/hello/world`".

Note : la commande go get se charge de récupérer des packages externes et de les intégrer dans votre workspace Go, par exemple :

```
$ go get gopkg.in/yaml.v1
```

Le résultat est un répertoire `$GOPATH/src/gopkg.in/yaml.v1` qui contient le package yaml.v1, et que vous pouvez dès lors référencer dans votre code.

"go build" et "go install"

L'intérêt d'avoir un workspace global et un chemin unique pour chaque package est de pouvoir appeler les commandes go de n'importe où. On a déjà utilisé "go build" pour compiler notre Hello World en lui passant le nom du fichier source; cette commande accepte aussi un chemin de package, auquel cas l'ensemble des sources du package est compilé. Dans ce dernier cas le chemin du package est relatif à `$GOPATH/src`.

Par convention le nom court du package est le dernier élément du chemin de ce package; c'est ce nom qui sera utilisé pour nommer le résultat de la compilation du package (binaire ou exécutable). Dans notre exemple ci-dessus, le nom court du package serait "world".

Tip : "go build" peut parfois avoir un comportement surprenant… Si on appelle "go build" sans argument, il considère le répertoire courant comme un package, et par conséquent génère un exécutable qui porte le nom du répertoire. Si on lui passe plusieurs fichiers `.go` en paramètres, l'exécutable portera le nom du premier fichier spécifié. On peut toujours utiliser "go build -o *xxx*" pour spécifier explicitement le nom du binaire généré "*xxx*".

Le résultat de "go build" est placé dans le répertoire courant, ce qui n'est pas forcément ce qu'on souhaite. En particulier, les packages qui sont des dépendances d'autres packages doivent être placés à l'endroit attendu de la structure du workspace, c'est à dire soit `bin` soit `pkg`; c'est le rôle de la commande "go install".

Exemple : Si on suppose l'existence d'un package `hello/world`, peu importe dans quel répertoire vous vous trouvez, la commande "`go install hello/world`" va compiler les sources qui se trouvent dans le package `hello/world` de votre workspace (désigné par `$GOPATH`) et placer le résultat à l'endroit standard.

Exécutable vs. bibliothèque

Vous avez noté que notre *Hello, World* commence par la déclaration "`package main`"; le package "main" est un nom de package spécial qui indique que le fichier doit générer un **exécutable** et contient une fonction `main()`.

Tous les fichiers source Go doivent commencer par une déclaration de package, qui spécifie à quel package appartient le fichier. A part le cas spécial du package "main", on utilisera en principe le nom court du package, qui correspond normalement au dernier sous-répertoire dans lequel se trouve le fichier source.

Ensuite on trouve les imports des packages que l'on va utiliser. Chaque package est référencé par son chemin (les packages système comme "fmt" ont des chemins courts).

Tentons maintenant d'externaliser la génération de la chaîne de caractères affichée par notre Hello World dans une bibliothèque :
1. Créez un répertoire `$GOPATH/src/lib/hellolib`
2. Dans ce répertoire créez un fichier `sayhello.go` qui contient ce qui suit :

```
package hellolib

func Sayhello(who string) string  {
        result := "Hello, " + who + " from hellolib!\n";
        return result;
}
```

3. Compilez et installez cette bibliothèque en tapant "`go install lib/hellolib`"

Si tout se passe bien, on observe que le résultat de la compilation est un fichier `hellolib.a` placé dans `$GOPATH/pkg/<archi>/lib/hellolib.a`.

Quelques remarques :
- Dans la déclaration des paramètres de la fonction Sayhello, le type (`string`) vient après le nom du paramètre (`who`). C'est ainsi pour toutes les déclarations de type en Go.
- La déclaration et l'initialisation d'une variable peuvent se faire simultanément avec l'opérateur "`:=`".
- Pour une bibliothèque, c'est à dire un package n'ayant pas de "main", si on utilise "`go build`" au lieu de "`go install`" aucun fichier n'est produit.
- Le nom du fichier `.a` produit par "`go install`" dépend uniquement du nom du sous-répertoire dans lequel se trouvent les sources (ici "`hellolib`"); le nom du ou des fichiers .go n'est pas pris en compte, pas plus que la déclaration "package" en tête des fichiers source.

Maintenant utilisons cette bibliothèque :
1. Modifiez votre fichier hello.go comme suit :

```
package main

import "fmt"
import "lib/hellolib"

func main() {
        msg := hellolib.Sayhello("Go")
        fmt.Printf(msg)
}
```

2. Compilez et installez l'exécutable en tapant "`go install hello`" (en supposant que le fichier se trouve dans `$GOPATH/src/hello`).

Si tout se passe bien, le nouvel exécutable hello a été généré sous `$GOPATH/bin`, et en l'exécutant on trouve bien ce qu'on attend :

```
$ hello
Hello, Go from hellolib!
```

Quelques remarques :
- Vous avez peut-être noté que la fonction `Sayhello` commence par une **majuscule** : ce n'est pas un hasard, pour Go cela signifie que la fonction est **exportée** et sera donc visible en-dehors du package. C'est assez surprenant quand on a l'habitude de langages où la casse n'a pas de sémantique associée, et où la visibilité des objets est déterminée par des mots-clés, mais ça a l'avantage de la concision.

- La déclaration "package" en tête du fichier sayhello.go définit l'identifiant du package; c'est cet identifiant qui est utilisé quand on référence la fonction "`hellolib.Sayhello`".

- Lors de l'import, on peut donner un alias au package en faisant précéder le chemin du package par un identifiant, par exemple `import h "lib/hellolib"` permet d'écrire `h.Sayhello` au lieu de `hellolib.Sayhello`. Ceci permet de résoudre des situations de conflit lorsqu'on importe plusieurs packages de même nom.

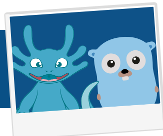
- On peut aussi grouper les imports sans répéter le mot-clé import à chaque ligne comme ceci :

```
import (
	"fmt"
	"lib/hellolib"
)
```

Chapitre 2

Les bases du langage

Les bases du langage



Go est un des nombreux langages dont la syntaxe est inspirée du C, avec notamment l'utilisation d'accolades `{` et `}` pour délimiter les blocs de code. On en a déjà rencontré quelques spécificités, comme l'opérateur "`:=`" pour le combiné déclaration/affectation de variable.

On peut citer aussi les spécificités suivantes :
- Le point-virgule "`;`" est un terminateur, mais il peut être omis dans la plupart des cas en fin de ligne,
- La déclaration de variables est introduite par le mot clé `var`. Comme on l'a déjà mentionné, le type suit l'identifiant. On peut aussi déclarer plusieurs variables du même type d'un coup, par exemple :

```
var x, y int
```

- Les variables peuvent être initialisées lors de leur déclaration, y compris pour les déclarations groupées, par exemple :

```
var url = "http://localhost"
var x, y int = 1, 2
```

- Dans le cas où on initialise les variables lors de leur déclaration, on peut omettre le type, par exemple :

```
var x, msg = 1, "Go go go!"
```

- On peut affecter une valeur à une variable qui n'a pas été déclarée, dans ce cas cela équivaut à une déclaration suivie d'une affectation :

```
msg = "Go go go!"
```

équivaut à :

```
var msg string
msg = "Go go go!"
```

- En l'absence d'initialisation, les variables sont toujours initialisées avec une valeur par défaut ("*zero value*"). Cette valeur est 0 pour les types numériques, `false` pour les booléens, la chaîne vide `''` pour les strings et `nil` pour les pointeurs.
- Les fonctions peuvent retourner deux résultats ou plus. La syntaxe est la suivante (exemple) :

```
func Swap(s1, s2 int) (int, int) {
        return s2, s1
}
```

Ici on déclare une fonction qui retourne une paire d'int. Le résultat de l'appel à cette fonction doit être affecté à une paire d'int comme suit :

```
x, y := Swap(1, 2)
```

- Le ou les résultats d'une fonction peuvent être nommés; les valeurs de retour sont traitées comme des variables dans la fonction, avec la possibilité supplémentaire d'écrire "`return`" seul, ce qui équivaut à retourner les variables de retour. **Cette pratique n'est pas encouragée** car elle nuit à la lisibilité du code.

Exemple :

```
func Swap(s1, s2 int) (r1, r2 int) {
        r1 := s2
        r2 := s1
        return //équivaut à return r1, r2
}
```

Les types en Go

Go connait nativement les types numériques suivants :
- Booléen : `bool`
- Entiers signés : `int8`, `int16`, `int32`, `int64` (8, 16, 32 ou 64 bits)
- Entiers non signés : `uint8`, `uint16`, `uint32`, `uint64` (8, 16, 32 ou 64 bits)
- Nombres en virgule flottante : `float32`, `float64`
- Nombres complexes : `complex64`, `complex128`

Il existe aussi des synonymes pour la lisibilité :
- `int` est synonyme selon la plateforme de `int32` ou `int64`
- de même `uint` est synonyme de `uint32` ou `uint64`
- `byte` est synonyme de `uint8` (un octet)
- `rune` est synonyme de `uint32` (un caractère unicode, on y reviendra dans la section ***Les subtilités des chaînes de caractères***)

Les types `complex64` et `complex128` représentent des nombres complexes, c'est à dire (rappel de maths) des nombres possédant une partie dite "réelle" et une partie dite "imaginaire". On peut les voir comme une simple paire de nombres en virgule flottante, mais ils possèdent une arithmétique propre. En Go on peut représenter un nombre imaginaire sous forme littérale en faisant suivre un nombre de la lettre "`i`".

On peut définir un nouveau type avec le mot-clé "type", par exemple :

```
        type Id uint32
```

Il peut s'agir d'un synonyme d'un type natif auquel on souhaite attacher une sémantique particulière ou bien d'une structure (dont nous reparlerons dans la section ***Structures***).

Conversions

Pour chaque type Go il existe une fonction de même nom qui convertit une valeur passée en paramètre vers la valeur correspondante du type cible (si la conversion est permise). Par exemple :

```
var x int = 1
var y float64 = float64(x)
```

Attention, il n'y a **pas de conversion implicite** en Go ! Le passage par une fonction de conversion est obligatoire. Par exemple ce qui suit est illégal :

```
var x float64 = 1.5
// cannot use x (type float64) as type int in assignment
var y int = x
```

Il faudra écrire :

```
var y int = int(x)
```

Itérations et conditionnelles

Go ne connaît qu'un seul opérateur d'itération : `for`. La syntaxe sera familière aux programmeurs C, Java etc. avec ses trois clauses séparées par des points-virgules :
- Initialisation : instructions exécutées **une fois avant la première itération**,
- Condition : expression booléenne évaluée **avant chaque itération**, la boucle s'arrête si elle est fausse,
- Post-itération : instructions exécutées **après chaque itération**.

A la différence des langages cités, ces trois composants ne doivent pas être entourés de parenthèses; en revanche les accolades du bloc de code qui suit sont obligatoires, même s'il ne consiste qu'en une instruction. Toute variable déclarée dans la partie initialisation n'est visible que dans ce bloc.

Par exemple :

```
        for i := 0; i < 10; i++ {
                fmt.Printf("%d\n", i)
        }
```

Si on ne spécifie pas les clauses d'initialisation et de post-itération, les points-virgules sont facultatifs, et du coup il ne reste que la condition : c'est l'équivalent d'un `while` en C.

Si on ne spécifie aucune clause, c'est l'équivalent de `for true` (boucle infinie); il faudra bien évidemment prévoir une condition de sortie (`break`).

Exemple :

```
f := 1
for {
	f += f
	fmt.Printf("%d\n", f)
	if f >= 100 {
		break
	}
}
```

Un `if` peut être vu comme un cas particulier de `for` où on exécute le bloc de code au maximum une fois. Comme pour un `for`, on peut avoir une clause d'initialisation qui sera exécutée avant l'évaluation de la condition; toute variable déclarée dans cette clause est visible dans le bloc de code associé (et uniquement dans ce bloc).

Exemple :

```
if d := rand.Intn(100); d < 10 {
	fmt.Println("REUSSI !")
}
```

Un `if` peut-être complété par un bloc `else`. Si des variables ont été déclarées dans la clause init du `if`, elles seront également visibles dans le bloc `else` :

```
if d := rand.Intn(100); d < 10 {
	fmt.Println("REUSSI !")
} else {
	fmt.Printf("PERDU : %d", d)
}
```

Si le bloc `else` consiste uniquement en une instruction `if`, les accolades sont facultatives, ce qui permet d'enchainer des tests `if` / `else if` / … `else` sans créer d'indentation de blocs :

```
if note >= 16 {
	fmt.Println("Mention Très bien")
} else if note >= 14 {
	fmt.Println("Mention bien")
} else if note >= 12 {
	fmt.Println("Mention assez bien")
}
```

Go dispose également de l'instruction `switch`. A la différence de langages comme C ou Java, dès qu'une branche a été exécutée, le switch ne continue pas avec la branche suivante, à moins de le spécifier explicitement avec l'instruction "`fallthrough`".

Comme pour un `if` on peut avoir une clause d'initialisation, par exemple :

```
switch d := time.Now().Weekday(); d {
case time.Sunday:
case time.Saturday:
	fmt.Println("WEEKEND !")
default:
	fmt.Printf("Pas weekend... %s\n", d.String())
}
```

Dans un `switch` Go il n'y a pas de contrainte sur le type des valeurs de chacun des `case`, ni sur leur caractère constant : un `switch` est simplement un raccourci d'écriture pour effectuer une suite de comparaisons entre un opérande évalué une fois (celui qui est dans le `switch`) et un ensemble d'autres opérandes (chacun des `case`) évalués à tour de rôle.

En d'autres termes :

```
switch v {
case v1:
	c1
case v2:
	c2
...
default:
	c0
}
```

est équivalent à :

```
if v == v1 {
	c1
} else if v == v2 {
	c2
}
...
} else {
	c0
}
```

Si on ajoute que l'opérande commun (`v`) est facultatif et vaut `true` par défaut, ceci permet de remplacer les cascades `if` / `else  if` / `else` par un `switch` utilisant des expressions booléennes pour les différentes valeurs des `case`.

```
        now := time.Now()
        switch {
        case now.Weekday() == time.Friday && now.Hour() >= 12:
                fmt.Println("Bientôt le weekend !")
        case now.Weekday() == time.Saturday
              || now.Weekday() == time.Sunday:
                fmt.Println("Weekend !!!")
        default:
                fmt.Println("Pas weekend...")
        }
```

Structures

Go possède la notion de structure : une structure Go est simplement une collection de champs nommés. L'utilisation la plus commune est de définir un type structuré via une déclaration similaire à ceci :

```
type Point struct {
    x, y float64
}
```

Un type structuré Go ne définit pas une classe au sens "orienté objet", dans la mesure où il ne définit pas les méthodes qu'on peut lui appliquer. Néanmoins on verra dans la section ***Méthodes*** que l'on peut déclarer des fonctions qui agissent sur une valeur de type en tant que "*type receveur*", ce qui en fait une notion proche de la notion d'objet.

On accède aux champs de la structure avec la notation "." habituelle dans de nombreux langages.

```
var p Point
fmt.Println(p.x, p.y)
```

On peut initialiser la structure statiquement comme ceci :

```
p := Point{1.0, 2.0}
```

On peut aussi spécifier les valeurs initiales des champs dans un ordre quelconque en spécifiant leur nom comme ceci :

```
p := Point{y: 2.0, x: 1.0}
```

Si l'on omet des champs, ils prendront leur valeur par défaut.

Si l'on n'initialise pas du tout la structure, sa valeur par défaut est telle que tous les champs ont leur valeur par défaut.

Tableaux et *slices*

Les tableaux sont manipulés dans Go d'une façon assez originale parmi les langages de programmation. La dimension du tableau fait partie de son type, de ce fait un tableau n'est pas redimensionnable; en revanche il existe la notion de "*slice*" c'est à dire de tranche de tableau, qu'on peut appréhender comme une vue dynamique s'appuyant sur un tableau sous-jacent.

La déclaration d'un **tableau** se fait de la façon suivante :

```
var x [10]int
```

Ceci déclare une variable x de type `[10]int`, c'est à dire "tableau de 10 `int`".

Un tableau peut être initialisé statiquement en utilisant une notation similaire à l'initialisation des structures :

```
arr := [5]int{1, 2, 3, 4, 5}
```

Une **slice** se déclare comme un tableau sans en préciser la dimension :

```
var s []int
```

On peut ensuite lui affecter un intervalle de tableau existant avec la notation suivante :

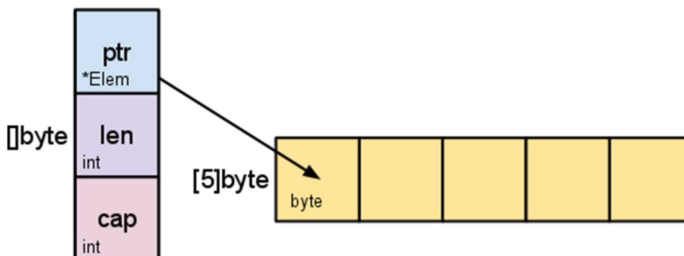
```
s := arr[0:2]
```

Ceci affecte à `s` une slice du tableau `arr` contenant les 2 éléments de `arr` à partir de l'indice 0.

Si l'on omet la borne inférieure et/ou supérieure, elles sont dans ce cas remplacées par respectivement 0 et le nombre d'éléments jusqu'au dernier indice du tableau :

```
s := arr[:2]  //2 éléments à partir du début du tableau
s := arr[1:3] //3 éléments à partir de l'indice 1 (inclus)
s := arr[2:]  //tous les éléments à partir de l'indice 2 (inclus)
```

Une slice possède en outre (en plus de sa longueur) une **capacité**, c'est à dire une longueur maximale. Celle-ci est limitée par le tableau sous-jacent.



Attention : une slice s'appuie toujours sur un tableau sous-jacent; la slice ne contient pas une copie des éléments du tableau mais référence directement le tableau. Tout changement via une slice affecte le tableau original !

On peut combiner l'initialisation statique d'un tableau et la création d'une slice en utilisant la même notation que l'initialisation statique de tableau mais sans spécifier la dimension :

```
s := []int{1, 2, 3, 4, 5}
```

Cette instruction effectue deux opérations distinctes :
- Création d'un tableau anonyme initialisé comme spécifié,
- Création d'une slice de ce tableau référençant tous les éléments.
C'est donc équivalent à écrire :

```
arr := [5]int{1, 2, 3, 4, 5}
s := arr[:]
```

Il est possible de créer dynamiquement une slice grâce à la fonction `make` : celle-ci effectue en une opération l'allocation d'un tableau (initialisé avec des valeurs par défaut) et retourne une slice de ce tableau; par exemple :

```
s := make([]int, 5, 5)
```

équivaut à :

```
var arr [5]int
var s = arr[:5]
```

On peut itérer simplement sur les éléments d'une slice (ou d'un tableau) grâce au mot-clé `range` comme suit:

```
for i, v := range s {
    fmt.Println("index:", i)
    fmt.Println("valeur:", v)
}
```

A chaque itération, `range` renseigne deux variables:
- L'indice de l'élément : `i`
- La valeur de l'élément : `v`
Si l'on n'est pas intéressé par l'index, on peut utiliser la pseudo-variable souligné "_", ce qui évite que le compilateur génère une erreur pour cause de variable non-utilisée.

Le type *map*

Une *map* est un dictionnaire qui associe une valeur à une clé; c'est l'équivalent d'un type `Map<K,V>` en Java par exemple.

Une map se déclare de la façon suivante :

```
var m map[K]V
```

Où K et V sont les types respectivement des clés et des valeurs, par exemple :

```
var values map[string]int
```

On peut initialiser une map statiquement d'une façon similaire à une struct :

```
	props := map[string]string{
		"lang": "go",
		"version": "1.9",
	}
```

Pour initialiser une map vide, on utilise `make` :

```
	props := make(map[string]string)
```

Pour insérer ou mettre à jour la valeur associée à une clé :

```
	props[cle] = valeur
```

Pour récupérer la valeur associée à une clé, on utilise la notation `m[cle]`; cette opération retourne une paire (`val, found`) où `found` est un booléen indiquant si la clé a été trouvée.

```
	val, found := prop[cle]
```

Note : Ceci permet de distinguer le cas où la clé existe et est associée à `nil`, du cas où la clé n'existe pas, ce que ne permet pas la fonction `Map.get()` de Java.

De façon similaire aux slices, on peut itérer facilement sur les paires (clé, valeur) grâce au mot-clé `range` comme suit :

```
for k, v := range props {
	fmt.Printf("%s -> %s\n", k, v)
}
```

Les subtilités des chaînes de caractères

La gestion des chaînes de caractères en Go diffère sensiblement des autres langages. A la base, une `string` en Go est simplement une séquence d'octets (une *slice* pour être exact), sans autre contrainte. Cependant, lorsqu'on manipule des `string`, notamment via les méthodes de la bibliothèque standard, Go interprète les chaînes de caractères comme étant **encodées en UTF-8**.

Si vous n'êtes pas familier avec UTF-8, il suffit de savoir qu'il s'agit d'un encodage qui permet de représenter une séquence de caractères *Unicode* sous forme d'une séquence d'octets, avec comme principale caractéristique que l'encodage d'un caractère Unicode se traduit par un nombre **variable** d'octets (de 1 à 4). La notion de caractère Unicode, ou plus précisément de *code point* selon la terminologie officielle, est représentée en Go par le type `rune`, qui est l'équivalent d'un `int32`.

La conséquence immédiate de ceci est qu'on ne peut pas utiliser l'indexation native pour accéder aux caractères, car ce sont les octets qui sont indexés et non pas les caractères. De même la fonction `len()` renvoie le nombre d'octets et non le nombre de caractères.

Par exemple le code suivant :

```
	str := "éèçà€"
	fmt.Printf("len=%d\n", len(str))
```

Fournit le résultat :

```
len=11
```

On note au passage que la chaîne est initialisée avec une valeur littérale qui contient bien 5 caractères... Ceci s'explique par le fait que le **code source Go est par définition encodé en UTF-8** et donc les chaînes littérales également.

Heureusement Go fournit des façons simples d'accéder aux caractères encodés dans une `string`. Ainsi le mot clé "`range`" permet d'itérer sur les caractères (c'est à dire les runes et non les octets) d'une string, comme suit :

```
	for i, r := range str {
		fmt.Printf("i=%d, r=%d, '%c' %+q\n", i, r, r, r)
	}
```

Produit le résultat :

```
i=0, r=233, 'é' '\u00e9'
i=2, r=232, 'è' '\u00e8'
i=4, r=231, 'ç' '\u00e7'
i=6, r=224, 'à' '\u00e0'
i=8, r=8364, '€' '\u20ac'
```

Comme on le voit dans l'exemple, les fonctions de formatage du package "fmt" sont également adaptées à l'encodage UTF-8.

Pour tous les autres besoins de manipulation de chaînes encodées en UTF-8, le package "`unicode/utf8`" fournit tous les outils nécessaires; par exemple `utf8.RuneCountInString()` fournit le véritable de nombre de caractères contenu dans une `string` :

```
fmt.Printf("RuneCountInString=%d\n", utf8.RuneCountInString(str))
```

```
RuneCountInString=5
```

Pointeurs

Go possède la notion de pointeur, comme en C ou C++, à la différence notable qu'on ne peut pas faire d'opérations arithmétiques avec les pointeurs, ce qui était une source d'erreurs extrêmement commune.

Contrairement à Java, une variable Go même lorsqu'elle est de type `struct` contient une **copie** de la valeur, et pas une référence. Si on souhaite manipuler une référence, on doit alors utiliser explicitement un pointeur.

La syntaxe est empruntée au C :
- une variable de type `*T` est un pointeur vers une valeur de type `T`,
- `&t` est l'adresse de la valeur de la variable `t`, c'est à dire un pointeur qui référence `t`,
- si `p` est de type `*T`, alors `*p` désigne la valeur référencée par `p`. De plus si `T` est une `struct`, alors on peut accéder aux champs de la structure via `(*p).f` ou bien simplement `p.f` (à comparer avec la notation C équivalente : `p->f`).

Les affectations de variables se font **toujours par copie**; de même le passage de paramètres à une fonction se fait par copie. Si on veut donner accès à l'objet original, il faudra alors passer à la fonction un pointeur vers cet objet.

Par exemple :

```
func set1(t time.Time) {
        t = time.Now()
}
var now time.Time
set1(now);
fmt.Println(now)
```

Si on exécute ce code, le résultat est que la valeur de la variable `now` n'a pas été changée par l'appel à `set1`, car dans ce cas lors de son exécution la variable `t` contenait une **copie** de now.

Si on modifie le code comme suit :

```
        func set2(t *time.Time) {
                *t = time.Now()
        }
        var now time.Time
        set2(&now)
        fmt.Println(now)
```

La fonction `set2` reçoit alors un **pointeur** vers l'objet `now` qui permet de modifier sa valeur.

Si une variable de type pointeur n'est pas initialisée explicitement, elle possède alors la valeur `nil`. Attention car alors il est possible de provoquer une erreur fatale ("invalid memory address or nil pointer dereference") si on essaye d'accéder à la valeur !

On peut allouer dynamiquement un espace-mémoire pour une structure avec le mot-clé `new`. L'appel à `new` prend un type en paramètre, et retourne un pointeur vers une valeur de ce type (initialisée avec la "valeur zéro" comme pour une variable), par exemple :

```
p := new(Point) // le type de p est *Point
```

Comme son homonyme Java, `new` alloue de l'espace mémoire dans la zone qu'on appelle le "tas" (*heap*), alors que les variables sont allouées sur la "pile" (*stack*) [1]. Tandis que l'espace mémoire alloué à une variable sur la pile est automatiquement libéré lorsque celle-ci est *déscopée* (sortie de fonction ou de bloc), l'espace alloué sur le tas sera désalloué par le ramasse-miettes (*garbage collector*) lorsque celui-ci déterminera qu'il n'est plus référencé. Par conséquent, comme pour tout langage à ramasse-miettes, l'allocation de mémoire via `new` doit être faite en connaissance de cause.

Type "fonction"

En Go les fonctions sont des objets typés au même titre que les autres et peuvent être affectées à des variables, passées en paramètre, etc. Ce sont des "first class citizens" du langage, ce qui rapproche Go d'un langage fonctionnel.

Une variable de type fonction est déclarée avec le mot-clé "`func`" suivi du profil de la fonction (arguments et type de retour) comme pour la déclaration de la fonction elle-même, mais sans le corps. Attention à la différence d'écriture avec l'implémentation :

---

[1] En principe, car le compilateur Go peut décider que l'espace mémoire associé à une variable doit être alloué sur la tas, notamment s'il n'est pas certain que celui-ci n'est pas référencé hors du scope de la variable; c'est le cas par exemple si la fonction retourne un pointeur vers cette variable. Ceci est entièrement transparent pour le développeur. Voir golang.org/doc/faq#stack_or_heap

`func f(args) typeret { corps }`	f est l'implémentation d'une fonction prenant comme arguments *args* et retournant une valeur de type *typeret*
`var f func(args) typeret`	f est une variable pouvant prendre comme valeur une fonction prenant comme arguments *args* et retournant une valeur de type *typeret*

Ainsi pour déclarer une variable f de type "fonction prenant deux `int` en argument et retournant un `int`" on écrira :

```
	var f func(x,y int) int
```

Si on suppose l'implémentation suivante :

```
	func somme(x,y int) int {
		return x+y
	}
```

On peut alors affecter à f cette implémentation comme suit :

```
	f = somme
```

On peut aussi affecter à une variable de type fonction une implémentation anonyme comme suit :

```
	f = func somme(x,y int) int {
		return x+y
	}
```

Les fonctions anonymes sont dans le scope de la fonction englobante, à la différence des fonctions nommées qui sont dans le scope du package.

L'invocation d'une fonction désignée par une variable se fait avec la même syntaxe que pour les fonctions nommées :

```
	res := f(3, 4)
```

Closures

Une *closure* est une fonction anonyme qui fait référence à des variables hors de son propre scope. Ceci permet en pratique de persister des données entre deux appels de la fonction, tout en s'assurant que seule cette fonction a accès à ces données. C'est en quelque sorte une fonction qui possède un état.

Comme exemple simple, on va créer une fonction qui enregistre le nombre de fois où elle a été appelée :

```
func newIncrementer() func() int {
	n := 0
	return func() int {
		n++
		return n;
	}
}
```

Ici `newIncrementer` est une fonction qui retourne une closure qui incrémente et retourne l'entier n. La variable n n'est pas dans le scope de la closure, mais elle est accédée par la closure et sa valeur est conservée d'un appel à l'autre.

Utilisation :

```
	inc := newIncrementer()
	fmt.Println(inc(), inc(), inc())
```

Résultat :

```
1 2 3
```

À noter que si `newIncrementer` est appelée plusieurs fois, chaque closure produite aura une copie différente de n ! Par exemple :

```
	inc1 := newIncrementer()
	inc2 := newIncrementer()

	fmt.Println(inc1(), inc1(), inc1())
	fmt.Println(inc2(), inc2(), inc2()
```

Résultat :

```
1 2 3
1 2 3
```

Méthodes

Dans un langage objet classique, les méthodes propres à un type sont associées directement à la classe qui définit ce type. Go n'est pas un véritable langage objet et n'a pas la notion de classe; il existe toutefois un mécanisme qui permet de définir des fonctions applicables à un type spécifique (dit "**type receveur**"), c'est ce que Go nomme "méthode".

La syntaxe est similaire à une fonction, en ajoutant la déclaration du type receveur entre parenthèses devant le nom de la fonction, par exemple :

```
type Rect struct {
        w , h float64
}

func (r Rect) Area() float64 {
        return r.w * r.h
}
```

L'appel se fait avec la notation pointée habituelle :

```
 area := r.Area()
```

Les méthodes sont en général définies sur des types-receveurs qui sont des structures, mais ce n'est pas obligatoire. Seule contrainte : les méthodes doivent être définies dans le **même package** que le type receveur.

Etant donné un type `T`, on peut définir des méthodes sur le type `T` (valeur de T) ou `*T` (pointeur vers une valeur de T). En pratique on peut appeler ces méthodes indifféremment sur une valeur de T ou sur un pointeur vers T, Go fera le nécessaire pour appliquer l'opérateur nécessaire.

Reprenons l'exemple du rectangle :

```
type Rect struct {
        w , h float64
}

func (r Rect) Area() float64 {
        return r.w * r.h
}

func (r *Rect) Area2() float64 {
        return r.w * r.h
}
```

```
        ...
```

On peut alors écrire :

```
	var r = Rect{1, 2}
	pr := &r

	r.Area()
	pr.Area() //utilisation implicite de *
	(*pr).Area()

	pr.Area2()
	r.Area2()  //utilisation implicite de &
	(&r).Area2()
```

Attention lors de la déclaration d'une méthode: si la méthode est déclarée sur le type receveur de base (pas le pointeur) elle recevra toujours une copie de la valeur; toute modification de cette valeur ne sera donc pas visible de l'appelant. Au contraire si on déclare la méthode sur le type receveur "pointeur", aucune copie n'a lieu et les éventuelles modifications sont visibles de l'appelant.

Pour illustrer supposons ces deux variantes de `setWidth` définies :

```
func (r Rect) setWidth1(width float64) {
	r.w = width
}

func (r *Rect) setWidth2(width float64) {
	r.w = width
}
```

L'appel à `setWidth1` n'affecte pas la valeur passée, au contraire de `setWidth2` :

```
	var r = Rect{5, 2}

	r.setWidth1(10)
	fmt.Printf("%f\n", r.w) //affiche 5.000000

	r.setWidth2(10)
	fmt.Printf("%f\n", r.w) //affiche 10.000000
```

Go et la composition

On sait déjà que Go ne permet pas l'héritage, cependant il possède un mécanisme facilitant la composition de structures : l'imbrication.

Dans une structure Go, tous les membres ne sont pas forcément nommés, il peut exister des membres dont on ne spécifie que le type (à condition que cela ne crée pas un conflit avec un membre nommé) : c'est ce qu'on appelle les **membres imbriqués**. Un membre imbriqué dans une structure se comporte comme un membre qui aurait le même nom que son type, à une différence près : les méthodes et champs du type imbriqué sont **promues** vers la structure contenante, c'est à dire que tout se passe comme s'ils étaient définis avec la structure contenante comme type receveur !

Un exemple permet de mieux comprendre. Imaginons une structure `Individu` avec nom et prénom ainsi qu'une méthode pour calculer le nom complet :

```
type Individu struct {
	Nom, Prenom string
}

func (i Individu) NomComplet() string {
	return fmt.Sprintf("%s %s", i.Prenom, i.Nom)
}
```

Maintenant ajoutons une structure qui associe un *login* à un Individu :

```
type User struct {
	Login string
	Individu
}
```

Ici le membre de type `Individu` n'est pas nommé; c'est un membre imbriqué. C'est là que la magie se produit : la méthode `NomComplet()` a été définie sur le type-receveur `Individu`, mais elle se trouve automatiquement **promue** et devient également accessible sur le type-receveur `User` !

Ainsi on peut écrire :

```
	individu := Individu{Nom:"Gérardin", Prenom:"Olivier"}
	login := User{Login:"oge", Individu: individu}
	fmt.Println(login.NomComplet())
```

En pratique tout se passe comme si on avait défini la méthode sur le type-receveur `User`

```
func (u User) NomComplet() string {
	return u.Individu.NomComplet()
}
```

...mais c'est inutile : Go s'en charge implicitement !

Du point de vue du développeur, la structure contenante se comporte effectivement comme si elle héritait de ses membres imbriqués, mais il est bon de ne pas perdre de vue le mécanisme qui permet ce comportement.

Interface

Une interface pour Go est simplement une collection de profils de fonctions.

Exemple :

```
type Surface interface {
        Area() float64
}
```

Ceci définit une interface appelée `Surface` qui comprend la fonction `Area` retournant un `float64`.

On dit qu'un type `T` implémente une interface `I` dès lors que **toutes les fonctions déclarées dans l'interface `I` sont implémentées** en tant que méthodes sur le type receveur `T`. L'implémentation d'une interface par un type est **implicite**, c'est à dire qu'elle n'est pas déterminé par une **déclaration** associée au type `T`. C'est ce qu'on appelle parfois *duck typing*[2].

Ceci contraste fortement avec des langages comme Java ou C++ où un type doit déclarer explicitement qu'il implémente une interface (`implements` en Java), et le compilateur vérifie que la déclaration est correcte, c'est à dire que toutes les méthodes sont implémentées.

Dans notre exemple, le simple fait d'avoir défini l'interface `Surface` comme ci-dessus fait que, sans déclaration supplémentaire, le type `Rect` défini précédemment implémente de *facto* cette interface, grâce au fait que la fonction `Area` existe sur ce type.

Une interface est un type; on peut affecter à une variable de ce type toute valeur d'un type qui implémente l'interface; par exemple étant donné ce qui précède il est valide d'écrire :

```
        r := Rect{1.5, 2.5}
        var s Surface = r
```

[2] "If it walks like a duck and it quacks like a duck, then it must be a duck." : si ça marche comme un canard et que ça caquette comme un canard, alors ça doit être un canard.

Une interface qui ne définit aucune méthode s'écrit `interface{}` et s'appelle l'**interface vide**. Par définition, n'importe quelle valeur implémente cette interface, et donc une variable de ce type pourra recevoir une valeur de n'importe quel type (à comparer au type `Object` de Java ou `void*` en C++).

Assertions et tests de type
Lorsqu'une variable est de type `interface`, elle contient en fait une valeur d'un type qui implémente cette interface. Pour accéder à cette valeur avec son type original, il faut utiliser une assertion de type comme suit :

```
var s Surface
...
var r Rect = s.(Rect) //suppose que s contient une valeur Rect
```

Si `s` ne contient pas une valeur de type `Rect`, alors l'opération échoue en déclenchant une erreur fatale (*panic*). Pour éviter ceci on peut en une seule opération **tester** si l'interface contient une valeur d'un type donné et récupérer cette valeur, comme suit :

```
r, ok := s.(Rect)
```

Si l'opération a réussi, alors `ok` vaudra `true`; sinon `ok` vaudra `false` et `r` contiendra la valeur zéro de son type.

On peut comparer ceci avec l'utilisation combinée en Java de `instanceof` suivi d'un *cast* :

```
if (x instanceof Y) {
        Y y = (Y) x;
        …
}
```

Tags et réflexion

Go permet d'obtenir programmatiquement des informations sur les types de données accédés, ceci au travers du package "*reflect*". Ceci permet d'implémenter un certain nombre de comportements dynamiques dépendants du modèle de données qui ne peuvent pas être décrits par des interfaces statiques, tels que la persistance de structures dans une base de données, etc. (voir chapitre ***Accès aux données***)

En outre, chaque champ d'une structure peut être associé à un tag contenant des "méta-informations" (accessibles via le mécanisme de réflexion). Ce système est comparable aux annotations qu'on trouve dans d'autres langages.

Le tag associé à un champ s'écrit entre *backquotes* `` après la déclaration du champ. Par convention, il s'agit d'une suite de paires *clé*:"*valeur*" séparées par des espaces, par exemple :

```
type User struct {
    Login string `bson:"login" json:"login" xml:"Login"`
}
```

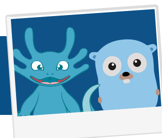
La clé désigne en principe le package concerné. Voici quelques exemples de packages qui utilisent les tags :

Clé	Package(s) concerné(s)
json	encoding/json, (mapping JSON par ex. Pour services REST)
xml	encoding/xml (mapping XML)
bson	gobson, mgo (mapping BSON pour MongoDB)
orm	github.com/astaxie/beego/orm (ORM)
gorm	github.com/jinzhu/gorm (ORM)
valid	github.com/asaskevich/govalidator (bibliothèque de validation)
datastore	appengine/datastore (Google App Engine platform, Datastore service)

Chapitre 3

Testing

Testing



Go propose nativement un support de test, même si celui-ci est minimaliste.

Pour en bénéficier :
- Créer un fichier dont le nom se termine par `_test.go`,
- Importer le module "`testing`",
- Y placer les fonctions de test qui doivent impérativement avoir le profil suivant :
  `func TestXxx(t *testing.T)` (Xxx sert à identifier le test)

Ensuite la commande "`go test`" va exécuter toutes les fonctions qui correspondent à ce profil dans les fichiers `*_test.go`.

La structure passée en paramètre aux fonctions de test permet de faire un certain nombres d'opérations comme du logging ou déclarer un test comme ayant échoué par exemple :

```
func TestArea(t *testing.T) {
	r := Rect{3.0, 4.0}
	area := r.Area()

	if area != 12.0 {
		t.Log("Area() : résultat %i, attendu %i", area, 12.0)
		t.Fail()
	}
}
```

Cependant, c'est à vous de valider le résultat des tests.

À noter que `t.Fail()` marque le test comme ayant échoué mais poursuit l'exécution, alors que `t.FailNow()` interrompt l'exécution (de la fonction courante) immédiatement.

Existent aussi :
- `t.SkipNow()` marque le test comme devant être sauté (attention si `t.Fail()` a été appelé avant, le test est considéré comme ayant échoué),
- `t.Error()` est équivalent à `t.Log(...)` suivi de `t.Fail()`.

Sur le même modèle que les tests, on peut écrire des fonctions de benchmarking avec le profil suivant :

```
func BenchmarkXxx(b *testing.B)
```

Cette fonction est supposée effectuer `b.N` fois l'appel au code à mesurer, donc dans le cas général :

```
	for n := 0; n < b.N; n++ {
		// code à benchmarker
	}
```

Exemple pour benchmarker la méthode Area() :

```
func BenchmarkArea(b *testing.B) {
	for n := 0; n < b.N; n++ {
		w := rand.Float64()
		h := rand.Float64()

		r := Rect{w, h}
		_ = r.Area()
	}
}
```

Le benchmarking est activé en appelant "`go test`" avec le flag "`-bench`"; Go va alors appeler les fonctions `BenchmarkXxx` plusieurs fois avec des valeurs différentes de N jusqu'à ce que la durée converge vers une valeur stable.

Attention : les fonctions de benchmark doivent être écrites de sorte à ce que le code à benchmarker soit d'un temps d'exécution constant; en particulier il ne doit pas dépendre du nombre d'itérations (B.N) ou d'autres paramètres, faute de quoi la convergence ne peut être assurée !

Comme pour les tests, les fonctions `Fail()`, `Error()`, `Log()`, etc. sont également disponibles sur `testing.B`.

Assertions

Pour les habitués de JUnit et frameworks de test, il existe une bibliothèque open source pour Go qui fournit un ensemble de mécanismes qui viennent compléter le mécanisme de base :

https://github.com/stretchr/testify

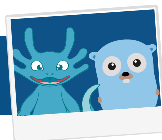
Cette bibliothèque permet en particulier de valider des assertions via des prédicats (similaire au package Assert de JUnit) et de faire du mocking.

Par exemple la fonction TestArea pourra s'écrire :

```
func TestArea(t *testing.T) {
	r := Rect{3.0, 4.0}
	area := r.Area()

	assert.Equal(t, area, 12.0)
}
```

Chapitre 4

Concurrence



Goroutines

En principe, l'appel à une fonction est **synchrone**, c'est à dire que l'exécution ne continue que lorsque la fonction a terminé de s'exécuter. On peut changer ce comportement en faisant précéder l'appel de la fonction du mot-clé "`go`"; dans ce cas la fonction est exécutée de façon **asynchrone** et l'exécution se poursuit sans attendre le retour de la fonction.

Concrètement la fonction s'exécute dans un fil d'exécution concurrent, appelé aussi "goroutine"[3] .

<u>**Exemple :**</u>

```
func rafale(id string) {
	for i := 0; i < 10; i++ {
		fmt.Printf("[%s]", id)
		time.Sleep(100)
	}
	fmt.Printf("[%s DONE]\n", id)
}

func main() {
	for i := 0; i < 5; i++ {
		go rafale(fmt.Sprintf("goroutine %d", i))
	}

	var input string
	fmt.Scanln(&input)
}
```

<u>**Note :**</u> la dernière ligne de la fonction main attend l'appui sur une touche; sans cette ligne la fonction main termine avant que les goroutines aient terminé et interrompt celles-ci sans en attendre la fin normale.

Si on exécute ce code, on note que les écritures provenant des différentes goroutines sont entremêlées et l'ordre de terminaison n'est jamais exactement le même, ce qui montre que leur exécution n'est pas séquentielle.

[3] Ce fil d'exécution est géré par Go et ne correspond pas forcément à un thread natif.

Channels

> "*Ne communiquez pas en partageant la mémoire, partagez la mémoire en communiquant*" - premier proverbe Go[4]

Pour communiquer entre Goroutines, Go possède la notion de "channel". Un channel est un objet de type FIFO (ou file de messages) qui permet l'envoi et la réception synchrone de messages **d'un type donné**.

La déclaration se fait avec le mot-clé `chan` (exemple pour un channel de strings) :

```
var c chan string
```

La création et l'initialisation d'un channel se font via la fonction `make`, qui prend en paramètres le type du channel et optionnellement une capacité :

```
c := make(chan string)
```

ou :

```
c := make(chan string, 10)
```

Pour envoyer un message vers le channel, on utilise l'opérateur `<-` comme suit :

```
c <- "msg"
```

Pour lire un message on utilise le même opérateur comme suit :

```
var msg string = <- c
```

La lecture est **bloquante** c'est à dire que l'exécution s'interrompt jusqu'à ce qu'un message soit disponible; le résultat est le contenu du message lu.

On peut spécifier la direction d'un channel en utilisant une des formes suivantes :

```
var c1 chan<- string // channel en écriture seulement
var c2 <-chan string // channel en lecture seulement
```

Bien entendu lors de sa création un channel est toujours bidirectionnel, mais si une fonction accepte un channel en paramètre on peut le déclarer sous une forme unidirectionnelle, ce qui contraindra la fonction à l'utiliser uniquement dans la direction spécifiée.

Voici un petit exemple qui publie toutes les secondes l'heure courante dans un channel (sous forme de string), alors qu'une goroutine lit les messages du channel et les affiche.

---

[4] Les *proverbes Go* sont une série de conseils issues d'une présentation de Rob Pike à Gopherfest 2015, qu'on peut retrouver en intégralité sur https://go-proverbs.github.io.

```
func receiver(c <-chan string) {
	for {
		var s string = <-c
		fmt.Println(s)
	}
}

func sender(c chan<- string) {
	for {
		now := time.Now()
		c <- now.String()
		time.Sleep(time.Second)
	}
}

func main() {
	var c chan string = make(chan string)
	go receiver(c)
	sender(c)
}
```

Que se passe-t-il si on veut traiter un message provenant d'un channel ou d'un autre ? Go dispose pour cela de l'instruction `select`, qui fonctionne comme un switch mais pour les channels, permettant ainsi le multiplexage de channels :

```
		select {
		case s := <-c1:
			fmt.Println("C1:", s)
		case s := <-c2:
			fmt.Println("C2:", s)
		}
```

L'instruction `select` bloque tant qu'aucun des channels n'a de message disponible; dès qu'un message devient disponible, la partie de code correspondant à ce channel est exécutée.

`select` permet aussi d'implémenter un *time-out* de la façon suivante :

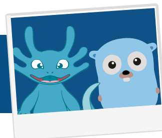
```
		select {
		case s := <-c:
			fmt.Println(s)
		case <-time.After(time.Second * 2):
			fmt.Println("timeout")
			return
		}
```

`time.After` crée un channel "à la volée" et y envoie le timestamp courant après un délai passé en paramètre. Si aucun message n'a été reçu sur le channel c avant ce délai, `time.After` enverra un message qui déclenchera la seconde branche du select (traitement du time-out).

Chapitre 5

Accès aux données

Accès aux données



Comme tout langage moderne et mûr, Go offre de nombreuses options pour persister et lire les données dans différents *backends*.

Il est aisé en Go d'implémenter le pattern DAO grâce à la notion d'interface, par exemple :

```
type PersonDAO interface {
	Save(*Person) error
	GetAll() []Person
	GetByID(id string) (*Person, error)
}
```

On peut ensuite implémenter cette interface pour différents backends et en version "mock" pour le test.

On va examiner le cas des bases de données MongoDB et SQL.

MongoDB

Sans aller jusqu'à dire que MongoDB est un choix "naturel" pour persister des données avec Go, les deux fonctionnent très bien ensemble grâce au package "mgo" (https://labix.org/mgo).

Le mapping entre les membres d'une `struct` et les champs du document Mongo associés se fait via le mécanisme de tags sur les champs (voir section ***Tags et réflexion***) en utilisant la clé "`bson`", par exemple :

```
type Person struct {
	Id        string `bson:"_id"`
	FirstName string `bson:"firstName"`
	LastName  string `bson:"lastName"`
	BirthDate t.Time `bson:"birthDate"`
}
```

Note : ce mapping est optionnel, par défaut mgo utilisera le nom du champ comme clé pour l'objet BSON persisté.

La connexion à une instance MongoDB se fait en utilisant `mgo.Dial(url)` ou une de ses variantes comme `mgo.DialWithTimeout`. Le résultat est un doublet `(*mgo.Session, error)`; en cas d'absence d'erreur la session peut être utilisée pour réaliser les opérations sur la base.

Typiquement on utilisera `session.Copy()` pour créer une nouvelle session avant d'effectuer des opérations; les sessions ainsi créées partagent les informations de connexion et le pool de connexions et peuvent être facilement passées à d'autres méthodes. Chaque session ainsi créée doit être clôturée avec la méthode `Close` pour

assurer la restitution correcte des ressources; cette opération est habituellement invoquée via la directive "`defer`" au début de la méthode, ce qui assure l'appel à `Close` quelle que soit l'issue de l'exécution de la méthode (à comparer avec un `try/catch` dans d'autres langages).

Ceci en tête, le DAO PersonDAO peut s'implémenter comme suit avec mgo :

```
import (
        "gopkg.in/mgo.v2"
        "gopkg.in/mgo.v2/bson"
)

type PersonDaoMongo struct {
        session *mgo.Session
}

const (
        collection      = "persons"
)

// Cet idiome fonctionne comme une assertion qui valide le fait
// que PersonDaoMongo implémente bien PersonDAO
var _ PersonDAO = (*PersonDaoMongo)(nil)

func NewPersonMongoDao(session *mgo.Session) PersonDAO {
        return &PersonDaoMongo {
                session: session,
        }
}

func (s *PersonDaoMongo) GetByID(id string) (*Person, error) {
        session := s.session.Copy()
        defer session.Close()

        person := Person{}
        c := session.DB("").C(collection)
        err := c.Find(bson.M{"_id": id}).One(&person)
        return &person, err
}


func (s *PersonDaoMongo) GetAll() []Person {
        session := s.session.Copy()
        defer session.Close()
```

```
	c := session.DB("").C(collection)
	persons := []Person{}
	c.Find(nil).All(&persons)
	return persons
}

func (s *PersonDaoMongo) Save(p *Person) error {
	session := s.session.Copy()
	defer session.Close()

	c := session.DB("").C(collection)
	_, err := c.UpsertId(p.Id, p)
	return err
}
```

SQL

Package "sql"

Pour accéder à une base de données SQL depuis Go, il existe une API officielle définie dans le package "`database/sql`" : https://github.com/golang/go/wiki/SQLInterface.

Cette interface "bas niveau" se positionne de façon similaire à JDBC dans le monde Java, et permet d'effectuer des opérations de base (connexion, exécution de requêtes et lecture de résultats, gestion des transactions, introspection, etc.) d'une façon identique quelle que soit la DB cible.

Pour accéder effectivement à une DB, il faudra compléter avec un package "*driver*" qui implémente cette API pour une base de données spécifique (de la même manière que pour utiliser JDBC il faut le driver spécifique à la DB) : https://github.com/golang/go/wiki/SQLDrivers.

La liste des bases de données pour lesquelles il existe un driver comprend les principales (liste non exhaustive) :
- Couchbase N1QL : https://github.com/couchbase/go_n1ql
- DB2 : https://bitbucket.org/phiggins/db2cli
- Firebird SQL : https://github.com/nakagami/firebirdsql
- MS SQL Server (pur go) : https://github.com/denisenkom/go-mssqldb
- **MySQL** : https://github.com/ziutek/mymysql
- **MySQL** : https://github.com/go-sql-driver/mysql
- ODBC : https://bitbucket.org/miquella/mgodbc

- ODBC : https://github.com/alexbrainman/odbc
- Oracle : https://github.com/mattn/go-oci8
- Oracle : https://github.com/rana/ora
- **Postgres (pur go) :** https://github.com/lib/pq
- Postgres (utilise cgo) : https://github.com/jbarham/gopgsqldriver
- Postgres (pur go) : https://github.com/jackc/pgx
- **SQLite (utilise cgo) :** https://github.com/mattn/go-sqlite3
- SQLite (utilise cgo) : https://github.com/gwenn/gosqlite
- SQLite (utilise cgo) : https://github.com/mxk/go-sqlite
- Sybase SQL Anywhere : https://github.com/a-palchikov/sqlago
- TDS (SQL Server et Sybase) (utilise cgo) : https://github.com/minus5/gofreetds

A noter qu'il existe une suite de tests certifiant la compatibilité des drivers Go SQL; les drivers en **gras** ci-dessus en font partie et la passent avec succès.

Certains de ces drivers utilisent "`cgo`", une technologie qui permet à Go d'utiliser des bibliothèques écrites en C. C'est un critère à prendre en compte si on souhaite rester en pur Go et faire de la cross-compilation.

Exemple d'utilisation du package sql (ici avec le driver MySQL `go-sql-driver/mysql`) :

Note : en principe le compilateur Go considère comme une erreur le fait d'importer un package sans l'utiliser. Cependant, on peut parfois vouloir importer un package pour bénéficier de ses effets de bord; c'est le cas des drivers SQL qui s'enregistrent auprès du système en tant que tels. Du coup pour permettre la compilation on utilise un "import muet" en spécifiant comme alias du package le caractère souligné "_".

```
import (
	"database/sql"
	_ "github.com/go-sql-driver/mysql"
	"testing"
	"log"
	"fmt"
)

func TestSQL(t *testing.T) {
	db, err := sql.Open("mysql", "root@(127.0.0.1:3306)/")
	if err := db.Ping(); err != nil {
		log.Fatal(err)
	}

	rows, err := db.Query(
		"SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES")
```

```
	if err != nil {
		log.Fatal(err)
	}
	for rows.Next() {
		var name string
		if err := rows.Scan(&name); err != nil {
			log.Fatal(err)
		}
		fmt.Println(name)
	}
	if err := rows.Err(); err != nil {
		log.Fatal(err)
	}
}
```

ORM

Gorm (http://gorm.io) est le package ORM le plus connu et le plus utilisé. Il gère la migration automatique de schéma (migration ascendante uniquement), les associations 1-1, 1-N, N-N, le polymorphisme, les transactions, etc. Son utilisation reste très simple pour le développeur.

Comme pour le package "`mgo`", le mapping se fait sur base d'une `struct` dont les membres sont potentiellement "annotés" avec le préfixe "`gorm`", par exemple :

```
type Pers struct {
	Id        uint      `gorm:"AUTO_INCREMENT"`
	FirstName string    `gorm:"type:varchar(20)"`
	LastName  string    `gorm:"not null"`
	BirthDate time.Time `gorm:"index"`
	IgnoreMe  bool      `gorm:"-"`
}
```

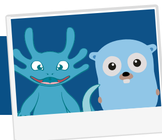
Si on active la migration automatique de schéma, Gorm crée automatiquement des tables sur base du nom de la `struct` (converti au pluriel); le nom des colonnes correspond au nom du champ converti en camel case. Si on n'utilise pas la migration automatique, il faudra spécifier le nom des colonnes à mapper.

Exemple simple d'utilisation avec MySQL :

```go
import (
	"github.com/jinzhu/gorm"
	_ "github.com/jinzhu/gorm/dialects/mysql"
	"testing"
	"time"
)

func TestGorm(t *testing.T) {
	db, err :=
	gorm.Open("mysql",
"root@(127.0.0.1:3306)/gorm?charset=utf8&parseTime=True&loc=Local")

	if err != nil {
		t.Fatal("failed to connect")
	}
	defer db.Close()

	// Mettre à jour le schéma
	db.AutoMigrate(&Pers{})

	toSave := Pers{
		FirstName: "Olivier",
		LastName:  "Gérardin",
		BirthDate: time.Now(),
	}

	// Sauvegarder une entité
	db.Create(&toSave)
	id := toSave.Id

	// récupérer via l'ID (fonctionne seulement avec les entiers)
	var pers Pers
	db.First(&pers, id)

	// récupérer via une clause WHERE
	db.First(&pers, "last_name = ?", "Gérardin")

	// récupérer au travers d'un exemple
	db.First(&pers, db.Where(&Pers{LastName:"Gérardin"}))

	// Supprimer
	db.Delete(&pers)
}
```

Chapitre 6

Web



Basics avec "http"

Go inclut un package "http" destiné à gérer les opérations HTTP aussi bien côté client que serveur. Ce package gère HTTP/1 mais également HTTP/2 (depuis la version 1.6), push (depuis la version 1.8), HTTPS, etc.

En quelques lignes il est extrêmement simple d'associer un handler à une URL et de démarrer un serveur HTTP :

```
import (
	"fmt"
	"net/http"
)

func main() {
	http.HandleFunc("/", func(w http.ResponseWriter,
					r *http.Request) {
		w.WriteHeader(http.StatusOK)
		w.Header().Set("Content-Type",
				"text/plain; charset=utf-8")
		fmt.Fprintln(w, "Golang HTTP server!")
	})

	http.ListenAndServe(":8081", nil)
}
```

Un simple browser web ou un appel à `curl` sur http://localhost:8081 permet de vérifier que cela fonctionne comme attendu :

```
$ curl http://localhost:8081
Golang HTTP server!
```

`http.HandleFunc` (ou son autre forme `http.Handle`) associe une fonction *handler* (respectivement une instance de `Handler`) à un pattern d'URL dans le multiplexeur par défaut. `ListenAndServe` démarre un serveur sur l'adresse spécifiée (`nil` indique ici d'utiliser le multiplexeur par défaut).

On peut également très facilement exécuter des requêtes HTTP avec `http.Get`, `http.Post`, `http.PostForm`, comme le montre l'exemple suivant :

```go
import (
	"io/ioutil"
	"log"
	"net/http"
)

func main() {
	resp, err := http.Get("http://localhost:8081")
	if err!= nil {
		log.Fatal("GET failed")
	}
	defer resp.Body.Close()
	body, err := ioutil.ReadAll(resp.Body)
	str := string(body)
	log.Println(str)
}
```

S'il est très simple de créer un serveur ou d'exécuter des requêtes clientes, le package "`http`" offre également la possibilité d'aller plus loin dans le contrôle des paramètres.

Ainsi, si l'on souhaite avoir la main sur les headers HTTP, la politique de suivi des redirections, etc. on peut utiliser l'objet `Client`, par exemple :

```go
	client := &http.Client{
		CheckRedirect: func(req *http.Request,
						via []*http.Request) error {
			return http.ErrUseLastResponse
		},
	}

	resp, err := client.Get("http://localhost:8081")
	// le reste comme précédemment
```

Pour plus de contrôle sur la couche transport, on peut aussi personnaliser l'objet `Transport`, par exemple :

```go
	tr := &http.Transport{
		MaxIdleConns:       10,
		IdleConnTimeout:    30 * time.Second,
		DisableCompression: true,
	}
	client := &http.Client{Transport: tr}
	// le reste comme précédemment
```

Autres options

Multiplexeurs

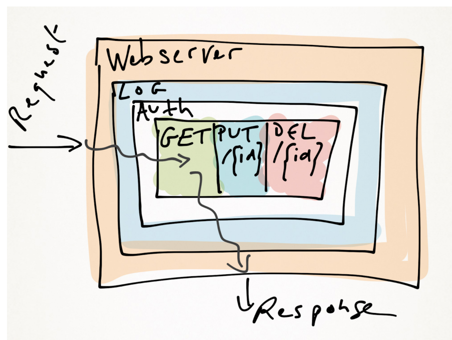
Le multiplexeur ou mux est un composant qui associe une requête HTTP à un handler, généralement sur base d'un pattern d'URL. Ceci permet de structurer un service web autour de templates d'URLs partiels, avec éventuellement des parties variables à l'intérieur, ce qui est essentiellement ce qu'on veut faire quand on écrit une API REST.

Si le multiplexeur par défaut (`http.ServeMux` du package "`http`") peut répondre à ce besoin, il existe de nombreux autres choix qui améliorent sensiblement les performances et/ou la flexibilité.

Package	Commentaires
Gorilla mux	**Signature des Handlers compatible ServeMux**
httprouter	**Les handlers prennent un troisième paramètre** `httprouter.Params`
bone	**Signature des Handlers compatible ServeMux via** `http.HandlerFunc(MyHandler)`
Goji	**Simplicité et concision.** **Signature des Handlers compatible ServeMux**

Middleware

En termes Go web le "middleware" est un ensemble de fonctionnalités appliquées transversalement à toutes les requêtes HTTP, comme le logging ou la sécurité (à comparer au mux qui fait correspondre UNE requête avec UN handler).



Si vous êtes familiers du monde Java/Spring, le mux est comparable à la notion de *filter* pour Spring MVC.

Negroni
Negroni est le plus utilisé des middlewares Go. Contrairement à d'autres middlewares qui intègrent la fonctionnalité de multiplexeur, Negroni fonctionne avec un mux au choix.

Negroni propose nativement une configuration "classic" qui prend en charge :
- `negroni.Recovery` - gestion des cas de panic et génération d'une réponse HTTP 500.
- `negroni.Logger` - Logging des requêtes/réponses.
- `negroni.Static` - Serveur de fichiers statique à partir du sous-répertoire "public".

Negroni s'intègre facilement avec le package "`http`", et du coup l'utilisation de Negroni nécessite peu de modifications de code; par exemple l'ajout de Negroni classic à l'exemple de base de la section ***Basics avec "http"*** nécessite très peu de changements :

```
import (
	"fmt"
	"net/http"
	"github.com/urfave/negroni"
)

func main() {
	mux := http.NewServeMux();
	mux.HandleFunc("/", func(w http.ResponseWriter,
	                                         r *http.Request) {
		w.WriteHeader(http.StatusOK)
		w.Header().Set("Content-Type",
			"text/plain; charset=utf-8")
		fmt.Fprintln(w, "Golang HTTP server!")
	})

	n := negroni.Classic()
	n.UseHandler(mux)
	http.ListenAndServe(":8081", n)
}
```

- On crée explicitement un mux via `NewServerMux` au lieu de configurer le mux par défaut,
- On associe ce mux à l'instance Negroni via `UseHandler`,
- Negroni implémente l'interface Handler, ce qui fait qu'on peut le passer en deuxième argument de `ListenAndServe` (on reconnaît typiquement là le pattern décorateur).

En complément ou en remplacement de la configuration Classic, il est très simple de paramétrer Negroni avec des middlewares additionnels via la fonction `Use()`. Un middleware est simplement une implémentation de l'interface `Handler` :

```
type Handler interface {
        ServeHTTP(rw http.ResponseWriter,
                        r *http.Request,
                        next http.HandlerFunc)
}
```

Les middlewares étant **chaînés**, c'est la responsabilité d'un middleware (s'il n'a pas déjà écrit dans `ResponseWriter`) de céder la main au middleware suivant en appelant `next(rw, r)`.

Il existe de nombreux middlewares compatibles avec ou requérant Negroni, en voici une liste loin d'être complète :

Package	Rôle
binding	Binding des requêtes HTTP dans des structs
cors	Support de Cross Origin Resource Sharing
gzip	Compression GZIP de la réponse HTTP
oauth2	Authentication OAuth 2.0
negroni-sessions	Gestion de session

A noter que Negroni a une documentation en français : https://github.com/urfave/negroni/blob/master/translations/README_fr_FR.md

Frameworks web

Il existe également des frameworks Go complets prenant en charge tous les aspects d'une application web, avec différentes approches, différentes performances et différentes facilités d'intégration. Comme cette liste ne peut être exhaustive, nous ne parlerons que des principaux.

Revel

Revel est un framework qui se veut complet, productif et performant. Il prend en charge

le routage, le parsing des paramètres, la validation, le templating, les tests, le cache, etc. En mode développement, il permet le rechargement à chaud.

Il est également extensible au travers d'une notion de filtre proche des Handlers de Negroni mais avec une interface différente.

Les plus :
- Complet, pas de dépendance externe

Les moins :
- Taille importante
- Mécanismes spécifiques (router, filtres, etc.)
- Pas de support natif MongoDB

Gin

L'argument essentiel de Gin est la performance. A l'inverse de Revel, Gin se veut minimaliste (et en cela plus conforme à la philosophie de Go). Gin est compatible au niveau API avec Martini, mais avec des performances nettement supérieures.

Les plus :
- Minimaliste
- Simple à apprendre (surtout si on connaît Martini)
- Rapide

Les moins :
- Minimaliste… nécessite l'utilisation de bibliothèques externes pour certaines fonctionnalités

Martini

Martini est en quelque sorte le framework web historique de Go et en cela il montre son âge; cependant il a inspiré d'autres frameworks comme Gin, grâce à son approche minimaliste et sa facilité d'intégration. Il est cité pour référence car il n'est plus activement maintenu.

Les plus :
- Minimaliste
- Très bien documenté

Les moins :
- Performances
- Plus activement maintenu

Tests web

Go inclut un package "`httptest`" (`net/http/httptest`) dédié aux tests web. Celui-ci permet, par exemple, de tester un Handler sans passer par la couche réseau, en simulant un appel HTTP. Pour cela :
- On crée une requête spécifique avec `httptest.NewRequest()`,
- On crée un `ResponseWriter` avec `httptest.NewRecorder()`. Il s'agit d'une implémentation "mock" de `ResponseWriter` qui enregistre les mutations pour pouvoir les inspecter par la suite,
- On appelle directement le Handler avec ces paramètres.

Exemple :

```
handler := func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "Golang HTTP server!")
}

testreq := httptest.NewRequest("GET",
                                "http://example.com/foo", nil)
recorder := httptest.NewRecorder()
handler(recorder, testreq)

resp := recorder.Result()
assert.Equal(t, http.StatusOK, resp.StatusCode)
defer resp.Body.Close()
bodyBytes, _ := ioutil.ReadAll(resp.Body)
bodyString := string(bodyBytes)
assert.Equal(t, "Golang HTTP server!", bodyString)
```

Chapitre 7

Conteneurisation

Conteneurisation

Go, en plus d'être le langage utilisé pour développer Docker, se prête particulièrement bien à la conteneurisation. En effet, sa compilation statique sans dépendance ni runtime à l'exécution facilite un déploiement dans une image légère.

Avec l'arrivée de la construction d'images Docker multi-étapes, il est à présent possible de compiler un binaire Go dans une première image de taille plus conséquente et de récupérer le résultat de cette compilation pour produire une image finale d'à peine quelques mégaoctets de plus que le binaire qu'elle contient. Un exemple de ce à quoi peut ressembler un tel DockerFile est donné en exemple ci-dessous :

```
# build step
FROM golang:1.10-alpine AS build-env

# GO and PATH env variables already set in golang image
# to reduce download time
RUN apk --no-cache add -U make git

# set the go path to import the source project
WORKDIR $GOPATH/src/github.com/Sfeir/monprojet
ADD . $GOPATH/src/github.com/Sfeir/monprojet

# In one command-line (for reduce memory usage purposes),
# we install the required software,
# we build the program with make
# we clean the system from all build dependencies
RUN make all && apk del make git && \
  rm -rf /go/pkg && \
  rm -rf /go/src

# final stage
FROM alpine:latest
# add root https certificates
RUN apk --no-cache add ca-certificates

WORKDIR /app
COPY --from=build-env /go/bin/todolist /app/

# by default, the exposed ports are 8020 (HTTP)
EXPOSE 8020
ENTRYPOINT ["/app/todolist"]
```

Conclusion

Un langage ne vaut que s'il est adopté par les développeurs. A cet égard on peut dire que Go réussit plutôt mieux que la plupart des autres langages; la dernière enquête annuelle "StackOverflow Developer Survey 2018" montre qu'il est 5ème parmi les langages les plus aimés avec 65,6%, et 3ème des langages les plus désirés avec 16,2%.

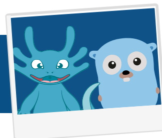
Que retiennent les développeurs qui se mettent au Go ? Ils apprécient sa syntaxe concise, son apprentissage rapide, sa vitesse de compilation et la simplicité du tooling, le typage statique… D'un autre côté, la concision de la syntaxe est parfois à la limite de l'aridité, la manipulation des pointeurs pas toujours évidente, l'absence de types paramétrés (généricité) ou d'héritage peut gêner ceux qui viennent d'un langage où ils existent, mais ces points sont rapidement oubliés.

Go est supporté par une communauté d'utilisateurs bouillonnante qui a produit un nombre impressionnant de bibliothèques, à tel point qu'on a parfois l'embarras du choix dans certains domaines !

Quel que soit l'avenir de Go, on peut déjà dire qu'il a rempli ses objectifs initiaux et au delà car il est devenu un langage majeur dans le paysage du développement en un temps assez bref. Il est déjà adopté par les acteurs majeurs du Cloud, Google évidemment mais aussi AWS, Cloudflare et Azure. Sans compter son utilisation dans un nombre d'outils croissant (Kubernetes, Docker, Hugo, Buffalo,...). La dynamique est en route et ne ralentit pas…

Bref, c'est le moment de sauter dans le train Go !

Aller plus loin



Livres

A cause de la (relative) jeunesse du langage, il n'existe encore que peu de livres traitant de Go, quasiment tous sont en anglais :

"**Programmer en Go : Pourquoi ? Comment ?**" par Rudy Rigot

"**The Go Programming Language**" par Alan A. A. Donovan (ISBN 978-0134190440)

"**Go in Action**" par Erica St Martin et William Kennedy (ISBN 978-1617291784)

"**Go in Practice**" par Matt Butcher et Matt Farina (ISBN 978-1633430075)

"**Introducing Go : Build Reliable, Scalable Programs**" par Caleb Doxsey (ISBN 978-1491941959)

Ressources online

Blogs et sites consacrés à Go

French GO - le site francophone des programmeurs Go
https://frenchgo.fr/

Le site officiel du langage Go - actualités, téléchargements, documentation
https://golang.org

Go cheatsheet - aide-mémoire compact sur la syntaxe de Go
https://devhints.io/go

GopherAcademy - nombreux articles et tutoriels sur Go
https://blog.gopheracademy.com/

GoDoc - centralise la documentation des packages Go
https://godoc.org/

Go By Example - des exemples pour toutes les spécificités de Go
https://gobyexample.com/

Learn Go Programming - série de tutoriels Go de niveau assez poussé
https://blog.learngoprogramming.com/

Articles et tutoriels

The Go type system for newcomers
https://rakyll.org/typesystem/

The Go Blog - Defer, Panic, and Recover
https://blog.golang.org/defer-panic-and-recover

Tooling Workshop - A workshop covering all the tools gophers use in their day to day life
https://github.com/campoy/go-tooling-workshop

Top Online Courses To Learn Go programming Language (Golang) For Beginners
https://goo.gl/DD1Cq4

Gopher Academy Blog - Using Go Templates
https://blog.gopheracademy.com/advent-2017/using-go-templates/

Let’s talk about logging
https://dave.cheney.net/2015/11/05/lets-talk-about-logging

Build a RESTful API in Go and MongoDB
https://dzone.com/articles/build-restful-api-in-go-and-mongodb

Outils et bibliothèques

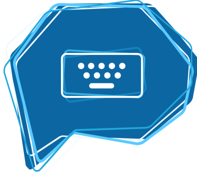
Go dep - documentation
https://golang.github.io/dep/docs/introduction.html

Vidéos

Go In 5 Minutes
https://www.youtube.com/c/go-in-5-minutes

Package main - Vlog d'Alex Pliutau à propos de programmation Go
https://www.youtube.com/c/packagemain

Just for func - Série de videos sur la programmation fonctionnelle en Go
https://www.youtube.com/c/justforfunc



PLAYOFFS

Chez SFEIR, chaque vendredi après-midi, c'est PlayOffs ! Il s'agit d'un processus de recrutement éxigeant au cours duquel le candidat fera connaissance avec jusqu'à cinq Sfeiriens différents autour de tests techniques. Sont alors évalués son savoir-être, son savoir-faire, son leadership, son potentiel et son empathie client. Plus que de simples entretiens nous faisons des PlayOffs de véritables échanges entre candidats et Sfeiriens à l'issue desquels nous sommes convaincus de vouloir travailler ensemble.
Rejoignez nous sur weare.sfeir.com.

Les Sfeiriens se retrouvent au moins une fois par mois lors des Soirées Techniques pour échanger autour d'un thème qui leur tient à coeur.
Les Sfeiriens qui le souhaitent peuvent y partager leurs dernières expériences ou recherches avec leurs collègues. C'est aussi une opportunité pour préparer leur présentation prévue pour un prochain meetup ou même une conférence ! Et c'est avec de bons bagels que la soirée continue pour échanger sur les sujets abordés autour d'une bière.

Meetup

Afin de favoriser l'essor des technologies et de leurs communautés, nous vous proposons d'accueillir vos meetups dans nos locaux et de financer le repas de la soirée. Pour cela, nous mettons à votre disposition notre salle Meetup ; elle est équipée pour accueillir dans les meilleurs conditions jusqu'à 70 personnes. Pour en profiter, il vous suffit de vous mettre en contact avec un Sfeirien qui veillera à ce que tout se déroule parfaitement.

À propos de SFEIR

SFEIR est avant tout une communauté façonnée par et pour des développeurs talentueux. Nous créons des applications de pointe et relevons avec nos clients leurs défis techniques les plus ambitieux. Nos domaines d'expertise sont au nombre de 9 : Front-end, Back, Mobile, Cloud, Data, Chatbots / Assistants, UX/UI Design, Internet of Things et DevOps.

Nous sommes fiers d'être développeurs. Nous partageons la même culture, ainsi que des valeurs qui nous sont chères : liberté, responsabilité, excellence, bienveillance et diversité.

Nous contribuons à l'épanouissement de chaque développeur/développeuse en partageant ce que nous apprenons et mettons tout en place pour aider tous les Sfeiriens à réaliser leur projet professionnel. Nous donnons des formations gratuites, écrivons des articles, organisons des meetups, et participons à de nombreuses conférences. SFEIR rassemble aujourd'hui plus de 450 développeurs à Lille, Luxembourg, Paris Strasbourg, Bordeaux et Nantes.

Envie de rejoindre notre équipe d'experts ? Rendez-vous sur weare.sfeir.com

Nos expertises

FRONT-END

BACK-END

MOBILE

CLOUD

DATA

IOT

DEVOPS

UX / UI

ASSISTANTS

N'hésitez pas à nous contacter :

commercial@sfeir.com

recrutement@sfeir.com

01 41 38 52 00

Remerciements

Nos plus sincères remerciements aux Sfeiriens·nes qui nous ont partagé leurs expériences ou contribué par des recherches approfondies à la production de ce livre blanc.

Un remerciement tout particulier à Olivier Gérardin, Sébastien Friess, Siegfried Ehret, Kadijatou Barry, Vladimir Kavaj.

www.sfeir.com